Факултет инжењерских наука Универзитета у Крагујевцу

Александар М. Михајловић

Тестирање софтвера: технике, методе и нивои тестирања са практичном применом

дипломски рад

Крагујевац, 2019.

Факултет инжењерских наука Универзитета у Крагујевцу

Назив студијског програма: Рачунарска техника и софтверско инжењерство

Ниво студија: Основне академске студије

Предмет: Софтверски инжењеринг 2

Број индекса: 586/2015

Александар М. Михајловић

Тестирање софтвера: технике, методе и нивои тестирања са практичном применом

дипломски рад

Комисија за преглед и одбрану:

1. **Др Велибор Исаиловић, доцент - ментор**
2. _____ Датум одбране:_____
3. _____ Оцена:_____

У оквиру завршног рада потребно је са теоријског аспекта обрадити тему тестирања софтвера:

- Технике тестирања (black box, white box, gray box),
- Методе тестирања (мануелно, аутоматско),
- Нивои тестирања (Unit, Integration, System, Security, Performance, Functional, Acceptance, User Interface, итд.),

Други део рада треба да обухвати практичну примену неког testing framework-а на примеру веб апликације. Примена подразумева писање одређеног броја тестова различитог нивоа за апликацију, са детаљима описа реализације тестова.

Препоручена литература:

[1] .....

[2] .....

[3] .....

Крагујевац, октобар 2019.

ментор:

Др Велибор Исаиловић, доцент

Садржај

Summary

The subject of this thesis is to show the basics of the software testing, specifically, the division of software testing based on technique, method, level, and type, and detailed explanation of each division. The aim of this paper is to explain all aspects of software testing as much as possible and, through practical examples, to demonstrate the benefit of testing each software by writing a number of tests.

Keywords: Software testing, test, bug, tester, test case

Резиме

Тема овог рада су основе тестирања софтвера, односно, подела тестирања софтвера на основу технике, методе, нивоа и типа, а такође и да се детаљно објасни свака подела. Циљ овог рада је да се детаљно објасне сви аспекти тестирања софтвера и да се кроз практичан пример, писањем одређеног броја тестова, покажу предности тестирања софтвера.

Кључне речи: Тестирање софтвера, тест, баг, тестер, тест случај

1. Увод

У данашње време, иако можда на први поглед не изгледа тако, наш живот у потпуности зависи од софтвера, јер се софтвер налази у великом броју система који користимо. Софтвер дефинише понашање мрежних рутера, банкарских мрежа, телефонских система, па чак и самог Интернета. Такође, софтвер је основна компонента која контролише рад сложених система попут авиона, свемисрких бродова, а може се наћи и у простијим системима као што су аутомобил, мобилни телефон или сат.

Веома добар пример последица лошег тестирања софтвера био би софтвер за управљање контролним торњевима на аеродромима. Ако софтвер закаже, да лош резултат, или се у тренутку сруши читав систем обавештавања пилота, дошло би до опште катастрофе, јер би се авиони сударали. Поставља се питање: како је могуће да је тестирани софтвер онда заказао? Разлог је једноставан: Програми су постали комплекснији. Рецимо, пре 20 година, за тестирање програма било је потребно две до три недеље. Данас, када су програми постали доста комлекснији, потребно је много више од две или три недеље да се тестирају. У неким случајевима чак и годину дана тестирања није довољно.

Наведени пример показује колико је значајно тестирање софтвера. Лоше или погрешно истестиран софтвер може веома лако довести до проблема и озбиљних последица. Нематеријанла и материјална штета, проузрокована радом недовољно истестираног софтвероа, у екстремним случајевима може бити несагледивих размера. Веома битан аспект тестирања је и безбедност података којима програм располаже, како не би дошло до злоупотребе од стране неовлашћених корисника софтвера. Из свега наведеног се може уочити колико је значајно свеобухватно тестирање софтвера пре његовог пуштања у рад у реалном окружењу.

У наставку су наведене дефиниције основних појмова који се користе приликом писања тестова и тестирања софтвера

Тест случај је спецификација улаза, услова, процедура тестирања и очекиваних излаза који дефинишу тест за одређену апликацију.

Тест је скуп улазних вредности, почетних услова извршавања, очекиваних резултата и стања у коме систем треба да остане након завршетка теста. Тест се развија са циљем да се испита одређено понашање програма или да се верификује исправност имплементације одређеног захтева.

Тестирање је поступак извршавања тестова над софтверским системом. Два основна циља тестирања су:

1. да се пронађу откази,
2. да се демонстрира исправно понашање система.

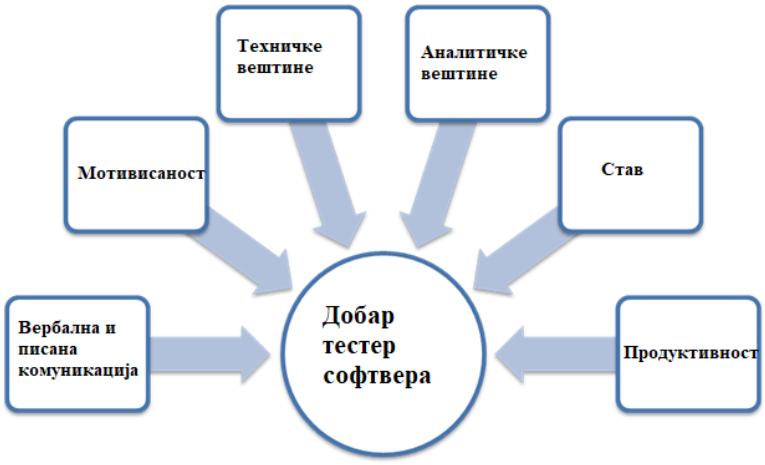
Појам тестирања софтвера је први пут користио научник Glenford J. Myers [1]1979. године. Он је јасно раздвојио појам тестирања од појма дебаговања. Тестирање софтвера преставља процес извршавања програма са циљем да се пронађу грешке унутар апликације. Ставке које сваки софтвер мора да потврди приликом тестирања су следеће:

- да ли задовољава све захтеве наручиоца софтвера,
- да ли реагује без појаве грешака на све врсте улаза,
- да ли је његово време извршавања прихватљиво,
- да ли је употребљив,
- да ли може бити инсталиран и да ли се може покренути у жељеним окружењима,
- да ли даје очекиване резултате.

Тестирање софтвера може почети чим програмер достави прву верзију програма (макар она само делимично радила). Број могућих тестова чак и за најједноставије програме је бесконачан. Тестирање софтвера за сваки програм је јединстено и увек се бирају тестови који ће дати најбоље резултате. Процес тестирања се може сматрати итеративним. Када се један баг[2] открије и поправи, може се појавити други, проблематичнији баг, који је такође потребно отклонити.

1.1 Тестер софтвера

Да би особа, која се никада није бавила тестирањем софтвера, постала успешан тестер, потребно је да поседује одређене техничке, а и нетехничке вештине (Слика 1.).



Слика 1. Битне особине сваког тестера [1]

[1] Амерички компјутерски научник, имао је утицај на развој микропроцесорске архитектуре.
[2] Популарни назив за софтверску грешку.

Кључне нетехниче вештине су следеће:

- **Аналитичке вештине**: Аналитичка вештина представља способност да се сваки комплексан софтверски систем подели на мање јединице како би био разумљив.
- **Комуникационе вештине**: Потребно је да тестер поседује добре комуникационе вештине, како би у случају када пронађе баг могао да објасни програмеру у чему је проблем. Такође мора да буде добро упознат са свим стратегијама, плановима и извештајима о баговима везаним за софтвер који тестира.
- **Организационе вештине**: Тестер мора добро да организује своје време, како не би дошло до кашњења испоруке програма клијенту.
- **Став**: Сваки тестер, мора да има став и тежњу ка перманентном учењу и усавршавању. У софтверској индустрији технологије константно напредују, тако да тестер мора да учи и да прати технолошке трендове.
- **Мотивисаност**: Добар тестер је увек мотивисан у свом послу. Увек се труди да уради свој посао како треба, квалитетно и са што мање, или чак без грешака.

Кључне техничке вештине које тестер треба да познаје су следеће:

- **Основно познавање база и SQL језика**: Сваки софтвер користи неку базу података. Те базе чувају доста информација, тако да у неким случајима тестер мора да приступи тим информацијама како би их проверио. Због тога мора да познаје најосновније SQL упите.
- **Основно познавање оперативних система**: Сваки софтвер мора да се развије на неком оперативном систему, тако да је веома значајно да тестер познаје оперативни систем на коме је израђена апликација, како би могао да примени тестове.
- **Познавање тест менаџмент алата (енгл. Test Management Tool)**: Сврха ових алата је да се чувају информације након сваког извршеног теста, тачније да се чувају извештаји извршених тестова и да се планирају следећи тестови.
- **Познавање алата за праћење дефекта (енгл. Defect Tracking tool)**: Алати за праћење служе како би у сваком тренутки сви чланови тима могли да знају где су пронађени дефекти у софтверу.
- **Познавање алата за аутоматизацију (енгл. Automation tool)**: У почетку сваки тестер почне да ради као мануелни тестер, али након неког времена природно је да пређе на аутоматизоване тестове, тако да му је потребно предзнање о алатима за аутоматизацију.

1.2 Основне поделе тестирања

Генерално посматрано постоје три категорије тестирања софтвера:

- Функционално тестирање
- Нефункционално тестирање или тестирање перформанси
- Одржавање (Регресија и одржавање)

Табела 1-категорије тестирања софтвера са њиховим примерима[3]

Категорија тестирања	**Типови тестирања**
Функционално тестирање	Unit тестирање
	Интеграционо тестирање
	Smoke
	UAT (Тестови од стране корисника)
	Локализација
	Глобализација
	Компатибилност
	...
Нефункционално тестирање	Перформанса
	Издржљивост
	Оптерећење
	Употребљивост
	Прилагодљивост
	...
Одржавање	Одржавање
	Регресија

Функционално тестирање је тестирање које проверава да ли свака функција ради у складу са датом спецификацијом. Ово тестирање углавном укључује тестирање методом црне кутије и не води рачуна о изворном коду апликације

Нефункционално тестирање има улогу да проверава нефункционалне аспекте апликације (перформансе, употребљивост, поузданост итд.). Главна улога му је да тестира спремност система према нефункционалним параметрима.

Тестови који припадају категорији одржавања су задужени за проверавање апликације након испоруке, као и у наредним годинама. Помоћу њих се исправљају и детектују грешке, сваки пут када се апликација ажурира или када јој се дода нова функционалност.

Према методи тестирања, тестирање софтвера се може поделити на:

- Мануелно
- Аутоматско

Код мануелног тестирања, тестери ручно извршавају тестове над апликацијом и притом не користе никакав алат. Мануелно тестирање представља најпримитивнију методу, али исто тако даје одличне резултате у проналажењу багова. Свака апликација се прво тестира мануелно, а затим може да се пређе на аутоматско тестирање.

Аутоматско тестирање представља коришћење алата за аутоматизацију. Сви алати за аутоматско тестирање раде по принципу *Record and Playback*. Постоје две фазе: фаза снимања (енгл. *record*) и фаза поновног пуштања (енгл. *playback*).

Према техници, тестирање се може поделити на:

- Тестирање методом црне кутије (енгл. *black box*)
- Тестирање методом беле кутије (енгл. *white box*)
- Тестирање методом сиве кутије (енгл. *gray box*)

Тестирање методом црне кутије је техника при којој тестери процењују функционалност програма који се тестира, без проласка кроз структуру кода.

Тестирање методом беле кутије је техника базирана на структури кода апликације. Код ове технике, за израду тестова, потребно је добро познавање система на коме се ради, као и добре програмерске вештине.

Тестирање методом сиве кутије је техника базирана на комбинацији методе црне и беле кутије. Тестер који користи ову методу мора да има приступ свим документима које се тичу апликације која се тестира, како би што боље направио тестове.

Према нивоу, тестирање се дели на:

- Јединично (енгл. *Unit*)
- Интеграционо (енгл. *Integration*)
- Системско (енгл. *System*)
- Прихватање (енгл. *Acceptance*)

Јединичним тестирањем се проверава да ли самостални модули кода апликације раде како је предвиђено, тако што се свака јединица апликације тестира одвојено.

Интеграционо тестирање је процес тестирања повезаности или преношења података између модула.

Системско тестирање је тестирање црном кутијом. У питању је процес тестирања целе апликације, како би се проверило да ли апликација ради и да ли задовољава све услове клијента. Такође, проверава се да ли за сваку комбинацију улаза даје жељене излазе.

Ниво прихватања представља презентацију апликације клијенту, како би апликација могла бити испоручена.

1.3 Седам основих принципа тестирања

Постоји седам основих принципа тестирања који се користе као смернице које би требало испоштовати у сваком пројекту. Принципи гласе**Error! Reference source not f ound.**:

1. Тестирање показује присуство дефеката – тестирање може да покаже да су дефекти присутни, али не може да докаже да у систему нема ниједног дефекта.
2. Исцрпно тестирање није могуће – тестирање свих могућих комбинација улаза и предуслова у пракси није могуће за било који систем.
3. Рано тестирање – како би се открили дефекти, тестирање треба да почне што је пре могуће.
4. Груписање дефекта (енгл. *defect clustering*) – тестирање треба фокусирати пропорцијално броју очекиваних и касније пронађених дефеката по модулима. Мањи број модула обично садржи већи број дефеката. Ти модули су обично најкритичнији и садрже имплементацију битних функционалности система.
5. Парадокс пестицида (енгл. *pesticide paradox*)– уколико се исти тестови понављају у свакој итерацији тестирања, на крају тај исти скуп тестова више неће моћи да

пронађе ниједан нови дефект. Тестове треба редовно прегледати и додавати нове тестове како би се избегао парадокс.

6. Тестирање зависи од контекста –различито се извршава у различитим контекстима.

7. Одсуство грешака не гарантује да систем ради како треба – проналажење и исправљање дефеката не помаже у случају да је систем неупотребљив или да не испуњава захтеве клијента.

2. Мануелно тестирање

Код ове врсте тестирања, тестер мануелно проверава рад апликације и упоређује са захтевима клијента. Главни циљ је да се осигура да је апликација без багова. Проверава се како апликација реагује приликом прелаза између екрана, да ли добро одговара при одређеним акцијама корисника и да ли даје задовољавајуће резултате. Мануелно тестирање графичког интерфејса је најчешћи тип овог облика тестирања. У великој мери, квалитет тестирања зависи од способности и искуства тестера, као и од његовог познавања апликације. Мануелно тестирање може бити прилично монотоно и напорно (понекад и фрустрирајуће), посебно када се изворни код апликације често мења.

Слика 2. Процес мануелног тестирања

Слика 2. описује процес мануелног тестирања. Кораци приликом тестирања су:

1. Прочитати и разумети документацију и записане захтеве апликације,
2. Написати тест случајеве који покривају све захтеве поменуте у документацији,
3. Прегледати тест случаје са тимом који ради на апликацији и са клијентом,
4. Извршити тестове над апликацијом,
5. Обавестити програмере о баговима,

6. Када су багови исправљени, поновити тест случајеве како би се проверило да ли је баг исправљен.

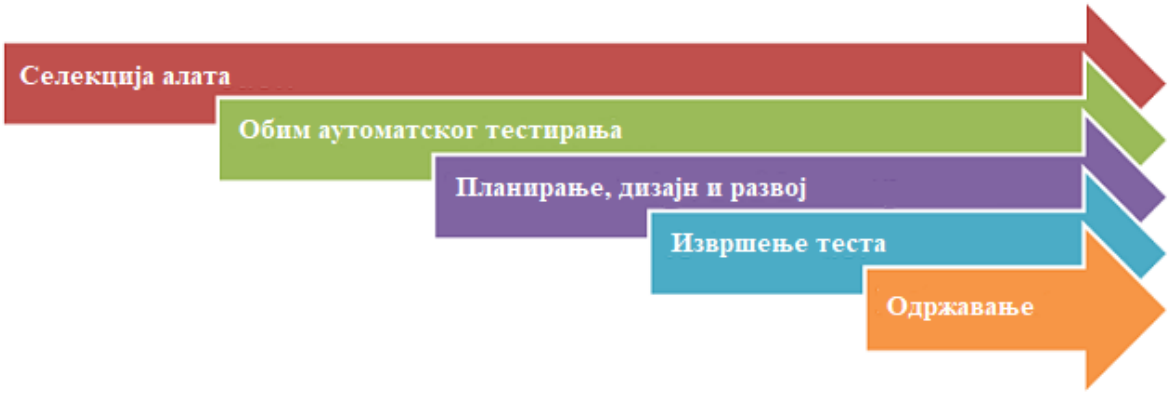
Такође, под мануелним тестирањем се води и случајно тестирање од стране одабране групе корисника. Случајно тестирање представља вид тестирања при коме се корисницима, који не знају изворни код апликације, даје та апликација на тестирање и циљ је да они коришћењем апликације случајно пронађу грешке у програму. Верзија апликације, коју одређена група људи тестирања на овакав начин, назива се бета верзија. Бета верзија се обично испоручује неколико недеља пре званичне испоруке софтвера.

3. Аутоматско тестирање

Аутоматско тестирање представља коришћење аутоматских алата ради извршавања тест случајева. Аутоматски алати могу унети тест податке, упоредити са очекиване и стварне резултате и на основу тога генерисати извештај теста. Ова врста тестирања захтева велика улагања у смислу новца и ресурса. Циљ ове методе тестирања је да се смањи број тест случајева који се извршавају мануелно. Међутим, циљ аутоматског тестирања није елиминација мануелног тестирања, јер пре аутоматског тестирања свакако је потребно мануелно тестирати апликацију.

Критеријуме које би требало поштовати, када је тестер у дилеми да ли да користи аутоматски тест над одређеним тест случајем су следећи:

- Тест случај је високог ризика,
- Тест случај је потребно понављати много пута,
- Тест случај је тешко извршити мануелно,
- Тест случај изискује много времена.



Слика 3. Процес аутоматског тестирања

На слици 3 је дат опис процеса аутоматског тестирања софтвера. Следи детаљнији опис фаза мануелног тестирања:

1. Селекција алата за тестирање највише зависи од комплексности апликације,
2. Дефинисањем обима аутоматског тестирања се описују детаљи апликације (комплекност, функционалност, количина података коју користи апликација, итд.),
3. Планирање, дизајн и развој је процес у коме се развија план и стратегија (бирање радног окружења, скрипти, припрема теста, итд.),
4. Изршавање тестова над апликацијом,
5. Ажурирање тестова, када дође до измене у коду апликације.

Неки од веома често коришћених алата за аутоматско тестирање су : Selenium, Ranorex Studio, mabl, Testim и QTP (MicroFocus UFT). Код аутоматског тестирања је веома важан избор адекватног алата, отклонити грешке у самом процесу тестирања и имати добар тим.

3.1 Алати за аутоматско тестирање

Selenium је алат који се користи за тестирање помоћу регресије. Алат је бесплатан и базиран је на принципу снимања и поновног пуштања. Подржава већину програмских језика као што су Пајтон, Јава, Руби, C# и има могућност извршавања више тестова одједном. Једина мана је што само подржава Mozzilu Firefox од свих интернет претраживача.

Ranorex Studio је алат који који подржава све врсте тестирања(енгл. *all-in-one*), као што су UI тестови, регресиони тестови, итд. Ranorex Studio је једноставан за коришћење и могу се писати тестови за интернет, десктоп и мобилне апликације.

mabl је алат који омогућава тестирање софтвера са што мање скрипти. Базиран је на моделима машинског учења за идентификацију багова. Тестови се аутоматски ажурирају приликом измене кода.

Testim је алат који користи вештачку интелигенцију како би убрзао извршавање и одржавање аутоматских тестова. Користи динамичке детекторе који уче приликом сваког извршавања теста и лако се прилагођавају променама кода. Такође, смањају време проведено на исправљању тестова. Резултати који се добијају су поприлично поуздани у већини случаја.

QTP (MicroFocus UFT) је алат који се највише користи за функционално тестирање и тестирање помоћу регресије. Принцип рада му је заснован тако што тестер пише тест случајеве у апликацији а затим помоћу њих обавља тестове.

4. Тестирање методом црне кутије

Тестирање црном кутијом је облик тестирања софтвера код кога унутрашња структура, дизајн и имплементација софтвера или софтверске јединице, која се тестира, нису познати тестеру. Овај облик тестирања се у потпуности базира на захтевима софтвера и његовим спецификацијама, тако да је спецификација апликације једини извор информација за дизајн и дефинисање тестова, јер импелементација и структура апликације нису познати. Пошто су познати само улаз и излаз, а оно што је између је непознато, отуда и назив црна кутија.

Слика 4. Принцип тестирања методом црне кутије

На слици 4 је представљен процес тестирања било ког софтверског система методом црне кутије. На пример, то може бити оперативни систем као што је *Windows*, или интернет страница као што је Гугл, па чак и нека сопствена апликација, тако да тестер може тестирати било шта од наведеног без познавања структуре. Рецимо, тестер може да испита апликацију рецимо кликом миша или уносом података са тастатуре и да упореди добијене резултате са очекиваним резултатима. Најчешће грешке које се проналазе методом црне кутије су сврстане у следеће категорије:

- Нетачне функционалности или функционалности које недостају,
- Грешке у интерфејсу,
- Грешке у структури података или грешке повезивања са базом,
- Грешке перформанси програма,
- Грешке при покретању или гашењу програма.

Тестирање методом црне кутије се може применити на свим нивоима тестирања, од јединичног па све до системског тетирања. Извршавање тестирања методом црне кутије се састоји из следећих корака:

- Проучавање захтева и спецификација посматраног система

- Тестер прво бира исправне улазе како би проверио да ли их систем правилно обрађује. Затим узима неке неисправне улазе како би проверио да ли систем детектује неправилности.
- Тестер прорачунава очекиване излазе за све улазе.
- Креирају се тест случајеви за одабране тест улазе.
- Извршавају се тест случајеви.
- Тестер упоређује стварне излазе са очекиваним излазима.
- Ако постоје грешке, оне се поправљају и систем се поново тестира

Постоје три типа тестирања црном кутијом:

- **Функционално тестирање** – овај тип тестирања црном кутијом је базиран на функционалним захтевима система.
- **Нефункционално тестирање** – овај тип тестирања црном кутијом не посматра функционалности ситема, већ нефунционалне захтеве система као што су перформансе, употребљивост, скалабилност.
- **Тестирање регресијом** – тестирање регресијом се извршава након исправљања кода, унапређивања кода или било које друге врсте одржавања како би се проверило да ли измењени код ремети остатак кода.

Једна од главних предности тестирања црном кутијом јесте да су тестови потпуно независни од имплементације програма, јер су базирани само на спецификацији система. Тако да су тестови употребљиви чак и у случају промене имплементације. Пошто се тестови пишу на основу спецификације система, тестови се могу развијати паралелно са развојем програма, што доводи до велике уштеде времена. Шта више, тестови се могу писати одмах након завршетка спецификације система.

Наравно, постоје и мане код ове технике, које се пре свега односе на то да постоји могућност да се део имплементације програма уопште не покрије тестом. Рецимо, то се дешава ако су у програм додате неке функционалности које се не налазе у спецификацији система. Такође, проблем код ове технике је и нејасна или некомплетна спецификација софтвера, што је у пракси врло честа појава, чиме је дизајн тестова знатно отежан.

Постоје три најбитније технике тестирања методом црне кутије:

1. Класа еквиваленције,
2. Анализа граничних вредности,

3. Табела одлучивања.

Класа еквиваленције је једна од основних техника црне кутије и користи се како би се оптимизовао број могућих тест случајева, при чему се задржава довољан број тестова за тестирање софтвера. Може се применити на било ком нивоу тестирања и најчешће је добар избор за почетно тестирање.

Анализа граничних вредности је тестирање фокусирано на граничним вредностима. Ова техника одређује да ли је одређени опсег вредности прихватљив за систем или не. Веома је корисно за смањивање броја тест случајева и најчешће се користи код система који за улазе узимају одређени опсег вредности. Ова врста тестирања је корисна за проналажење грешака зато што су грешке програмера у обради граничних случајева веома честе (рецимо када се уместо >= стави >).

Табела одлучивања, за разлику од класе еквиваленције и анализе граничних вредности који се фокусирају на кориснички интерфејс, се фокусира на пословну логику и правила дефинисана у спецификацији и представља одличан начин за тестирање комбинација различитих улаза. Табела одлучивања функционише тако што се сваки случај и његова последица смешта у матрицу, тако да свака ћелија има јединствену комбинацију.

Избор алата за тестирање методом црне кутије увелико зависи од типа тестирања. За функционалне и регресионе тестове највише се користе QTP и Selenium, а за нефункционалне тестове користе се LoadRunner и Jmeter.

5. Тестирање методом беле кутије

Тестирање методом беле кутије се дефинише као тестирање структуре, дизајна и кода апликације. Код ове методе тестирања, код је битан и тестер има приступ коду. Примаран фокус је на верификацији улаза и излаза апликације, али исто тако и на побољшању дизајна и употребљивости и унапређивању сигурности. Циљ тестирања код ове методе је да се изврше све програмске структуре, као и структуре података у програму. Спецификација се не проверава приликом примене методе беле кутије. Ова метода тестирања се може применити на свим нивоима тестирања (јединично, интеграционо, структурно) и најчешће се примењује након метода црне кутије.

Како утврдити да ли је систем довољно тестиран? Појам покривености тестирања (енгл. *coverage*) представља меру до које је одређени софтвер или његова компонента истестирана, у процентуалном смислу. Уколико покривеност није 100%, потребно је проширити скуп тестова новим тестовима са циљем да се повећа проценат покривености. Покривеност се рачуна по формули:

$$\text{Покривеност} = \frac{\text{Број ставки које су извршене}}{\text{Укупан број ставки}} \text{ x } 100\%$$

Битно је назначити да када је покривеност једнака 100% не значи да је софтвер 100% тестиран, већ само значи да су сви делови кода покривени. Метода беле кутије проверава следеће у коду:

- Унутрашње сигурносне пропусте,
- Структуру кода,
- Ток специфичних улаза у коду,
- Очекиване излазе,
- Функционалност условних петљи,
- Појединачно тестирање сваког исказа, објекта и функције.

Извршавање тестирања методом беле кутије прати два једноставна корака:

1. **Разумевање изворног кода** – Прва ствар коју тестер мора да уради јесте да проучи и разуме изворни код апликације. Пошто тестирање методом беле кутије захтева познавање изворног кода апликације, тестер мора да познаје и програмске језике који се користе у апликацији. Сигурност је један од главних задатака приликом тестирања софтвера, тако да тестер мора да нађе сваки сигурносни баг и да спречи нападе малициозним програмима или злоупотребили апликацију.

2. **Креирање тест случаја и извршавање** – Други корак тестирања меодом беле кутије је тестирање изворног кода апликације. Један начин је писање додатног кода како би се тестирао код апликације. Тестер мора да осмисли мале тестове за сваки процес апликације. Ово наравно захтева да тестер добро познаје изворни код апликације и због тога често ово ради и сам програмер. Други начини је мануелно тестирање, пробно тестирање и коришћење алата за тестирање.

Пример тестирања методом беле кутије дат је на Слици 4.

```
Printme (int a, int b) {
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }
```

Слика 4. Пример теста методом беле кутије

Као што је напоменуто, циљ тестирања методом беле кутије је да се провере све гране, петље и искази у коду. На Слици 4, тест случајеви које би требало анализирати овом методом су да је A = 1, B = 1 и A = -1, B = -3. Тиме би се проверило да ли програм улази и у *if* део петље и у *else* део петље.

Постоје три типа тестирања методом беле кутије:

- **Јединично тестирање** – Јединично тестирање се извршава над сваком јединицом или блоком кода у време писања кода. Обично га извршава програмер. Багови који се пронађу у овој фази су јефтини и лако се исправљају
- **Тестирање "цурења[3]" меморије** – "Цурење" меморије је главни узрок спорог рада апликације. Тестер који је експерт у проналажењу овог проблема је кључан у случајевима где се јавља успореност апликације.
- **Тестирање пробојности** – Код овог типа тестер или програмер има потпуну информацију о изворном коду, детаљне информације о мрежи, о IP адресама које управљају апликацијом и све информације о серверима који одржавају

[3]"Цурење" меморије је ситуација када се у извршавању програма изгуби информација о локацији динамчки алоцијаног блока меморије. У том случају програм више не може да ослободи тај блок меморије и он остаје изгубљен.

апликацију. Циљ је да се нападне код из различитих углова како би се откриле сигурносне опасности, ако постоје.

- **Тестирање мутације** – Ова врста тестирања се углавном користи како би се откриле најбоље технике које се могу искористити у случају да је потребно проширење апликације.

Једна од главних предности тестирања белом кутијом је боља оптимизација кода јер се проналазе сакривене грешке у коду. Тестови за ову методу се веома лако аутоматизују, што олакшава само тестирање и смањује утрошак времена. Такође, значајна предност је што је овај вид тестирања темељан, јер се пролази кроз сваки сегмент кода. Битно је напоменути и да тестирање може да почне и без корисничког интерфејса.

Наравно, постоје и мане код ове технике, а главна је да тестови код ове методе могу да буду веома комплексни и скупи. Такође, да би се тестирало овом методом, захтевају се људи са великим искуством и знањем, како би били у стању да разумеју имплементацију и код. И на крају, тестирање методом беле кутије понекад одузме много времена, јер велике апликације захтевају детаљно тестирање, које изискује доста времена.

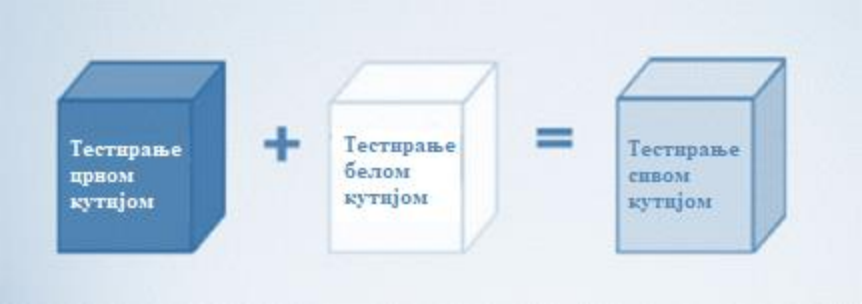
Технике тестирања методом беле кутије су типа анализе покривености кода. Анализа покривености кода елиминише пропусте у тест случајевима и идентификује области програма које не покривају тест случајеви. Када се детектују те области, тестер креира нов тест случај за тај део кода који није истестиран. Постоје две најбитније технике тестирања методом беле кутије и то су:

1. **Покривеност исказа** – ова техника захтева да се истестира сваки исказ унутар кода бар једном током процеса тестирања софтвера.
2. **Покривеност одлука –** ова техника проверава сваку одлуку (if-else и остале условне петље) унутар изворног кода апликације.

Алати који се највише користе приликом тестирања методом беле кутије су: Veracode, JSUnit, JUnit и EclEmma.

6. Тестирање методом сиве кутије

Метода сиве кутије је метода код које је за тестирање апликације потребно делимично знање изворног кода и структуре апликације. Помоћу ове методе се обично проналазе грешке везане за веб системе. Метода сиве кутије је комбинација метода црне и беле кутије. Код тестирања црном кутијом структура кода је непозната, код тестирања белом кутијом је структура кода позната, а код тестирања сивом кутијом је структура делимично позната (Слика 5.).



Слика 5. Тестирање методом сиве кутије

Метода сиве кутија има могућност да тестира обе стране апликације, и презентациони слој и слој у коме се налази изворни код. Пример би рецимо био ако се тестира нека функционалност вебсајта, као што је линковање и ако тестер наиђе на неки проблем са линковањем, он може да оде у HTML код, исправи грешку и поново провери у реалном времену.

Предност коришћења методе сиве кутије је зато што користи добре стране метода црне и беле кутије. Једна од предности је да се користе особине програмера као и тестера што побољшава квалитет програма. Такође, скраћује дуг процес тестирања функционалних и нефункционалних типова, што даје доста слободног времена програмеру да се фокусира на исправљању грешка. И на крају тестирање се извршава из угла корисника, а не из угла дизајнера.

Технике које се користе приликом тестирања сивом кутијом су:

- **Матрично тестирање** – ова техника тестирања укључује све променљиве које постоје унутар апликације.
- **Тестирање регресијом –** проверава да ли промене у прошлој верзији програма утичу на нову верзију. Извршава се тако што се користе стратегије: тестирај све поново, тестирај поново ризичне случај, итд.
- **Тестирање ортогоналним низом или OAT** – Пружа максималну покривеност кода са минималним бројем тест случајева.

- **Тестирање по шаблону** – ова техника се ослања на податке о грешкама у апликацији које су се догађале током развоја. За разлику од тестирања црном кутијом, тестирање сивом кутијом истражује код и покушава да разуме због чега се грешка догодила.

Кораци који се прате приликом тестирања методом сиве кутије су следећи:

1. Идентификовање улаза и излаза,
2. Идентификовање главних делова кода,
3. Идентификовање подфункција,
4. Реализација идентификованих улаза и излаза за подфункције,
5. Извршавање тест случајева над подфункцијама,
6. Провера резултата над подфункцијама,
7. Понављање корака 3 и 6 за све остале подфункције,
- Понављање корака 5 и 6 за све остале подфункције.

7. Нивои тестирања

Тестирање по нивоу је процес софтверског тестирања где се свака јединица или компонента софтвера тестира. Циљ овог тестирања је да се провери да ли свака компонента система задовољава прописана правила. Постоје различити нивои тестирања који проверавају понашање и перформансе софтвера. Ти тестови су дизајнирани да препознају делове кода које недостају. Постоје четири основна нивоа тестирања и они су приказани на Слици 6.

Слика 6. Нивои тестирања

Сваки ниво тестирања има специфичну улогу. Поред ових основних постоје и остали типови нивоа тестирања, а то су:

- Тестирање регресијом,
- “Buddy” тестирање,
- Алфа тестирање,
- Бета тестирање.

7.1 Јединично тестирање

Јединично тестирање представља тестирање код кога се тестира свака компонента софтвера. Ово тестирање се извршава над апликацијама током процеса развоја, тј. кодирања. Циљ јединичног тестирања је да изолује део кода и провери његова тачност. Јединично тестирање обично извршава сам програмер и представља први ниво тестирања, после кога следи интеграционо тестирање. Такође, ово тестирање спада у методу тестирања белом кутијом, јер се испитује изворни код апликације. Иако би требало да га извршава сам програмер, у већини случаја, због недостатка времена, овај ниво тестирања некада раде и тестери. Изостављање неког од јединичних тестова може довести до високих трошкова током системског и интеграционог тестирања. Правилно и темељно јединично тестирање чува време и новац приликом израде апликације. Предности коришћења јединичног тестирања су следеће:

- Јединичним тестирањем се поправљају багови на време, што смањује губитке,
- Помаже програмерима да боље разумеју суштину кода како би могли лакше да врше измене,
- Добри јединични тестови могу да служе као пројектна документација,
- Могу се тестирати делови пројекта иако цео пројекат није завршен.

Битно је назначити да јединични тестови нису идеални. Рецимо, овим тестирањем није могуће пронаћи сваки баг унутар програма, чак ни код тривијалних кодова. Такође, јединично тестирање не може да пронађе грешке које су интеграционог типа.

Јединично тестирање се може извршити аутоматски или мануелно, с тим да се аутоматско тестирање чешће користи. Аутоматско јединично тестирање функционише тако што програмер изабере део кода који жели да тестира, напише тест који треба да испита тај код, убаци у програм, изврши тест, погледа резултате и након тога узима следећи део кода. Алати који се често користе при јединичном тестирању су:

- **Junit** – бесплатан алат који користи Јава програмски језик. Овај алат прво тестира улазне податке, па тек онда тестира цео код.
- **Nunit** – окружење које се користи за све .net језике. Беспалатан алат који има могућност мануелног писања скрипти, такође скрипте могу радити паралелно.
- **PHPUnit** – алат за тестирање апликација које су писане у PHP-у. Узима мали део кода који се назива јединица и тестира сваки посебно.

7.2 Интеграционо тестирање

Интеграционо тестирање се дефинише као ниво тестирања код кога се појединачни модули и јединице спајају и тестирају заједно као целина. Пошто интеграционо тестирање следи након јединичног тестирања, компоненте које улазе у интеграционо тестирање су већ успешно прошле јединично тестирање. Истестиране компоненте се групишу у целине над којима се извршавају посебно припремљени тестови. Разлог због чега се поново тестирају компоненте унутар интеграционог тестирања, је чињеница да те компоненте први пут раде заједно, тако да их је потребно истестирати као групу, јер постоји могућност да се неке грешке тек тада испоље. Интеграциони тестови углавном имају фокус на корисничком окружењу и току података, то јест на размени информација између модула. Постоје два приступа у интеграционом тестирању:

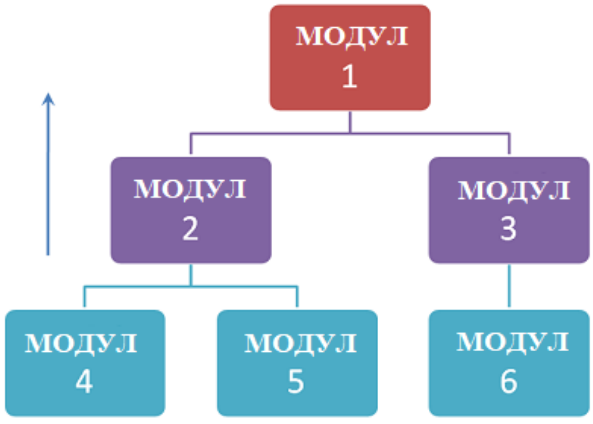
- **Приступ великог праска** – код овог приступа све компоненте се спајају и тестирају одједном. Најпогоднији је за мале системе, али постоји могућност лоших резултата зато што се сви модули тестирају одједном, а не постепено.

- **Постепени приступ** – код постепеног приступа спајају се два или више модула који имају заједничке карактеристике. По обављеном тестирању та два модула, додаје се следећи модул и тако све док се не додају и истестирају сви модули.

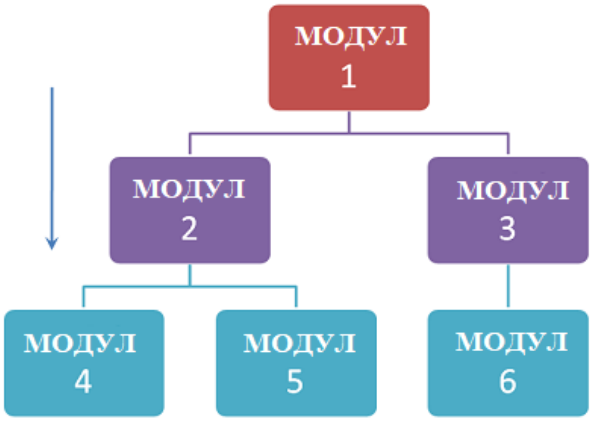
    - Одоздо на горе (Слика 7.) – сваки модул на нижем нивоу се тестира са модулом који је за ниво виши.

    - Одозго на доле (Слика 8.) – ток тестирања је такав да прати ток самог софтвера, почиње се од модула највишег нивоа и комбинује се са модулима који су ниво ниже.

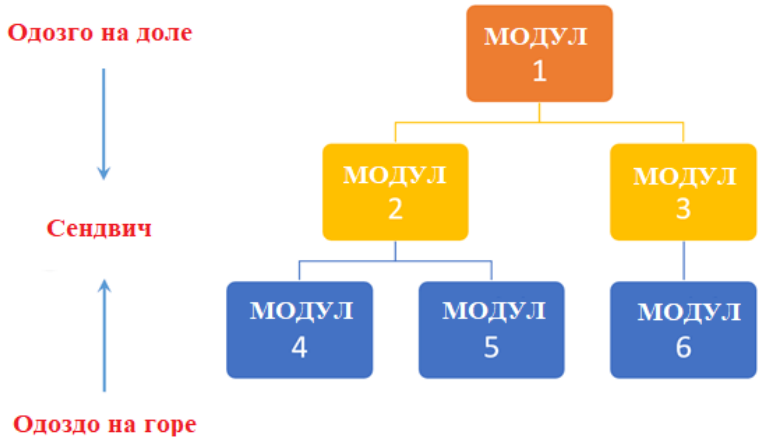
    - Приступ сендвича (Слика 9.) – је комбинација приступа одозго на доле и одоздо на горе. Модули на највишем нивоу се тестирају са модулима на најнижем нивоу.



Слика 7. Приступ одоздо на горе



Слика 8. Приступ одозго на доле



Слика 9. Сендвич приступ

Кораци извршавања интеграционог тестирања су следећи:

- Припремити план интеграционог тестирања
- Дизајнирати тест случајеве и скрипте
- Извршити тест случајеве и забележити сваки баг
- Исправити баг и поново тестирати
- Трећа и четврта тачка се понављају док се све компоненте (модули) апликације не испитају

7.3 Системско тестирање

Системско тестирање представља тестирање целог програма, након успешно извршене комплетне интеграције. Циљ системског тестирања је да се се истестира цео програм. Системско тестирање спада у технику тестирања црном кутијом, због тога што се не тестирају делови и рад изворног кода, већ цео софтвер. За разлику од јединичног тестирања које најчешће обављају програмери и интеграционог које спада у домен тестера, системско тестирање најчешће спроводе најискуснији тестери. Фокусира се на комплетном интегрисању апликације, укључујући и екстерне периферије и друге софтверске системе, како би се проверило да ли све компоненте раде исправно као целина, заједно са апликацијом. То у пракси значи да се апликација тестира у реалном сценарију употребе, са базом података, мрежом, свим хардвером и другим системима. Битно је напоменути да се тестирање обавља из перспективе корисника, уз детаљну проверу сваког улаза и излаза.

Описано тестирање је функционалног типа, поред тога системско тестирање може бити и нефункционалног типа, рецимо да се испитују перформансе система, комуникација са другим системима, скалабилност, поузданост и слично.

Постоји преко 50 типова системског тестирања, али од тих 50, седам типова системског тестирања је највише у употреби:

1. **Тест употребљивости** –фокус овог теста је кориснички интерфејс, да ли корисник може лако да користи апликацију и да ли апликација испуњава своје циљеве.

2. **Тест оптерећења** – неопходна је ова врста теста како би се проверило да ли апликација може да поднесе оптерећења у реалном времену.

3. **Тестирање регресијом** – ова врста системског теста проверава да ли су промене у коду довеле до неких нових багова. Такође је ту да осигура да се не појављују стари багови.

4. **Тест опоравка** – ова врста теста се извршава како би се демонстрирало да је систем поуздан, поверљив и да може лако да се опорави после системског пада.

5. **Тест миграције** – проверава да ли апликација без проблема може да се прабаци из старијих оперативних система у новије.

6. **Функционални тест** – суштина овог теста је анализа о могућим функцијама које недостају. Тестери обично смишљају листу свих функционалности које би апликација требало да има и онда упоређују са функцијама која апликација стварно има.

Тест хардвера/софтвера – код овог тестирања, тестер сву пажњу усмерава на интеракцију између хардвера и софтвера током тестирања апликације.

7.4 Алфа тестирање

Алфа тестирање је део теста прихватања од стране корисника (енгл. UAT – *User acceptance testing*) и спада у једну од најчешће коришћених стратегија у развоју софтвера. Главна сврха је да се пронађу сви потенцијални багови пре испоруке софтвера клијентима на бета тестирање. Тестирање се врши у лабораторијским условима на локацији фирме која производи саму апликацију, где тестери, који притом нису радили на тестирању те апликације, употребом различитих метода, црне, беле или сиве кутије, проверавају сваку функцију апликације. Алфа тестирање обично изводи тим независних тестера у оквиру компаније. Независан тим се узима због објективности.

Ово тестирање се врши на самом крају развоја софтвера, како би се све пронађене грешке исправиле пре испоруке клијенту и како би што мање проблема било при бета тестирању од стране клијента.

7.5 Бета тестирање

Бета тестирање је тестирање које изводи клијент. Бета тестирање извршавају стварни корисници апликације у стварним условима. Бета верзија апликације се испоручује ограниченом броју стварних корисника који је инсталирају и користе у реалним условима. Циљ је да корисници употребом апликације открију потенцијалне грешке или пропусте из своје перспективе. Као резултат тестирања, корисници дају своју оцену о квалитету производа.

Бета тестирањем се смањује ризик од неуспеха софтвера. То је финално тестирање пре испоруке софтвера свим корисницима. Постоје две врсте бета тестирања:

- **Затворена бета** – апликација се испоручује одабраној групи појединаца, који је тестирају.
- **Отворена бета** – апликација је доступна већој групи или чак целој јавности. Корисници пријављују сваку грешку коју уоче, а врло често и дају предлоге за које мисле да би побољшали апликацију.

8. Примена тестирања над апликацијом

Сви поменути тестови биће реализовати над реалном апликацијом. Апликација је израђена у Laravel окружењу (frontend и backend), коришћењем MySQL базе података. Апликација представља систем који користе магационери. Приступ апликацији имају запослени у магацину. Поред запослених приступ има и регистровани добављач, који доставља производе магацину. Запослени и добављач имају различите могућности у апликацији. Сваки запослени има могућност да види све информације о производу који се налази у магацину, да дода нови производ који је пристигао у магацин, да избрише или измени податке о већ постојећем производу, да дода новог добављача и да наручи производ кога нема на стању од постојећих добављача. Апликација чува податке о сваком производу: назив производа, опис, цена, произвођач и количина сваког производа који се налази у магацину. Паковање и складиштење производа у бази података је базирано на QR коду[4] и сваки производ има свој јединствен QR код, помоћу кога се може приступити том конкретном производу. Са друге стране, добављач има могућност да види производе који се траже од њега и да прихвати и пошаље те производе магационеру који их је тражио. Када добављач потврди пошиљку, прави се и фактура, коју добија магационер.

Што се реализације тиче, апликација је урађена у Laravel бесплатном развојном окружењу, базираном на PHP програмском језику. Дизајнирано је за развој веб апликација базираних на MVC[5] архитектури (енгл. *Model-View-Controller*). Релативно је младо окружење, али у кратком временском периоду је постало популарано међу програмерима. Тренутно представља једно од најпопуларних PHP окружења захваљујући стабилности, једноставности, лакоћи оптимизације апликације, јасној структури и брзом учитавању веб страна. У себи има уграђене функције за делове кода који се понављају у свакој апликацији, који олакшавају процес кодирања. Ти делови су аутентификација корисника, сесије, кеширање, управљање током контроле. Главне компоненте Laravel окружења су Composer, Middleware и Blade.[4]

Composer је алат који омогућава декларисање библиотека које пројекат користи, а такође води рачуна о њиховој инсталацији и ажурирању.

Middleware односно међусофтвер, је софтвер који ради између апликације и мреже. Пружа механизам за филтрирање HTTP захтева који долазе апликацији.

Blade је алат за приказе (енгл. *view*), односно оно што корисник види. За разлику од других алата, овај алат не забрањује коришћење обичног PHP кода у приказима. Увелико умањује посао израде *front-end-a* и пружа бољу функционалност. Главна предност коришћења blade-a је то што може да наслеђује друге приказе, што програмеру олакшава дизајн. Сви прикази користе *BootStrapp* CSS развојно окружење.

[4] QR код је матрични код, који када се дешифрује садржи сакривену поруку или број, слично бар коду.
[5] MVC је образац софтверске архитектуре који описује начин како да се структуира апликација.

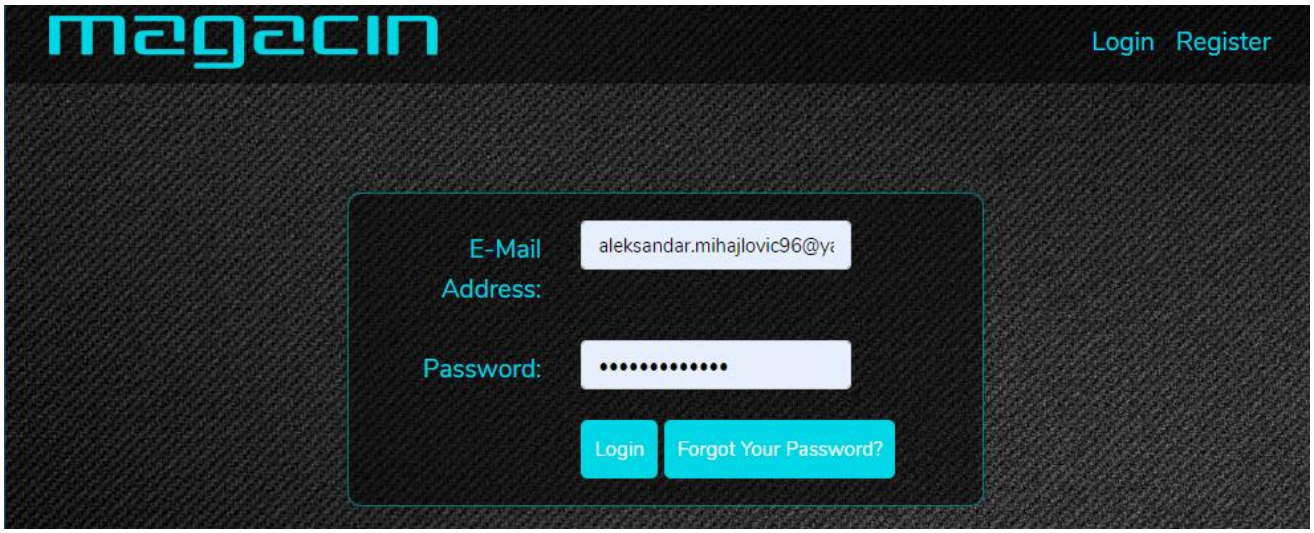
8.1 Пример мануелног тестирања

Као што је речено, код мануелног тестирања тестер, ручно тестира софтвер, то јест кликом миштa или уносом података са тастатуре. Као пример мануелног тестирања, у наставку ће се бити тестирана функционалност пријаве корисника. Тестер пре свега мора да анализира све могућности. Дакле, пошто се тестира пријава корисника, могућности које тестер може да испита су следеће:

- Провера пријаве постојећег корисника са тачним подацима (шифром и e-mail-ом),
- Провера пријаве корисника који не постоји у бази података апликације,
- Провера пријаве постојећег корисника са нетачном шифром.

Након исписаних свих могућности, тестер почиње да пише тест случај, за сваку од ових могућности. Рецимо тест случај за проверу пријаве постојећег корисника са тачним подацима би изгледао овако:

1. Отићи на страницу за пријаву корисника (Слика 10.),
2. Укуцати податке за логовање,
3. Проценити резултат теста (Слика 11.).



Слика 10. Страница за пријаву корисника

Слика 11. Почетна страна након успешне пријаве корисника

Пошто је корисник успешно пријављен и приказана му је почетна страна апликације, може се закључити да је тест прошао и да функционалност пријаве постојећег корисника апликације ради. Након провере свих набројаних могућности, тестер мора да испише детаљан извештај, у коме ће се налазити информације о сваком урађеном тесту. Пример таквог извештаја се налази на Слици 12.

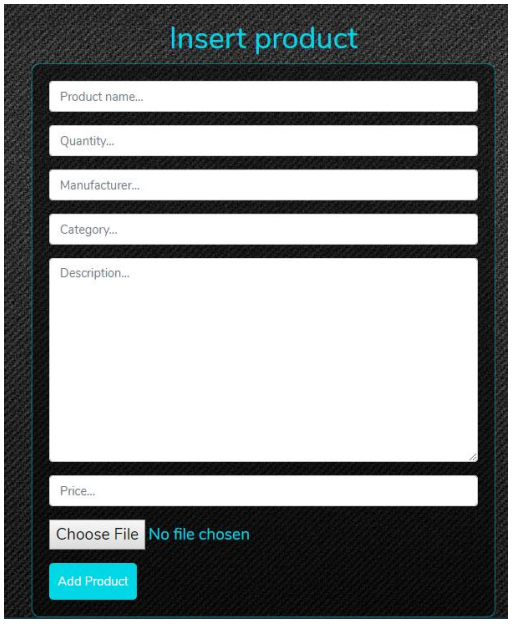
Назив пројекта:		Магацин електронских производа				**Број тест случаја**	**3**
Име и презиме тестера:		Александар Михајловић				**Прошао**	2
Датум почетка тестирања:		23.09.2019				**Пао**	1
Врста тестирања:		Мануелно тестирање - Тестирање логин стране					
Број тест случаја	**Сврха тест случаја**	**Предуслов**	**Кораци извршења теста**	**Улазни подаци**	**Очекивани излаз**	**Стварни излаз**	**Статус теста**
1	Провера функционалности логовања корисника са нетачном шифром	Корисник мора бити регистрован	1. Отићи на веб страницу; 2. Укуцати податке за логовање	1. http//nesto 2. Корисничко име: admin ; Шифра: 1234567	1. Корисник треба да има могућност да приступи страници за логовање; 2. Корисник треба да буде обавештен да је шифра погрешна;	1. Корисник има приступ страници за логовање; 2. Корисник не може да се улогује са погрешном шифром.	Прошао
2	Провера функционалности логовања корисника који не постоји	Корисник не треба да буде регистрован	1. Отићи на веб страницу; 2. Укуцати податке за логовање	1. хттп; 2. Корисничко име: проба; Шифра: 123	1. Корисник треба да има могућност да приступи страници за логовање; 2. Корисник треба да буде обавештен да корисник не	1. Корисник има приступ страници за логовање 2. Корисник није обавештен да корисник не постоји, већ да је погрешан унос.	Пао
3	Провера функционалности логовања корисника који постоји	Корисник мора бити регистрован	1. Отићи на веб страницу 2. Укуцати податке за логовање	1. http//nesto 2. Корисничко име: admin ; Шифра: 1234567	1. Корисник треба да има могућност да приступи страници за логовање; 2. Кориснику треба да се отвори почетна страна	1. Корисник има приступ страници за логовање; 2. Кориснику је отворена почетна страна сајта.	Прошао

Слика 12. Извештај о мануелном тестирању

Мануелно тестирање апликације је завршено када се провери сваки визуелни део апликације. Након потпуног мануелног тестирања апликације следи аутоматско тестирање.

8.2 Пример тестирања црном кутијом

Пример тестирања црном кутијом биће приказан на “Add Product” делу апликације . Тестирање црном кутијом је облик тестирања софтвера код кога унутрашња структура, дизајн и имплементација софтвера или софтверске јединице, која се тестира, нису познати тестеру. Дакле, једина информација коју тестер добија, јесте да он уносом података о производу треба да унесе тај производ у базу података. Тестер добија само форму са слике 13 и на основу ње треба да испита сваки могући сценарио у коме би могла да се појави грешка.

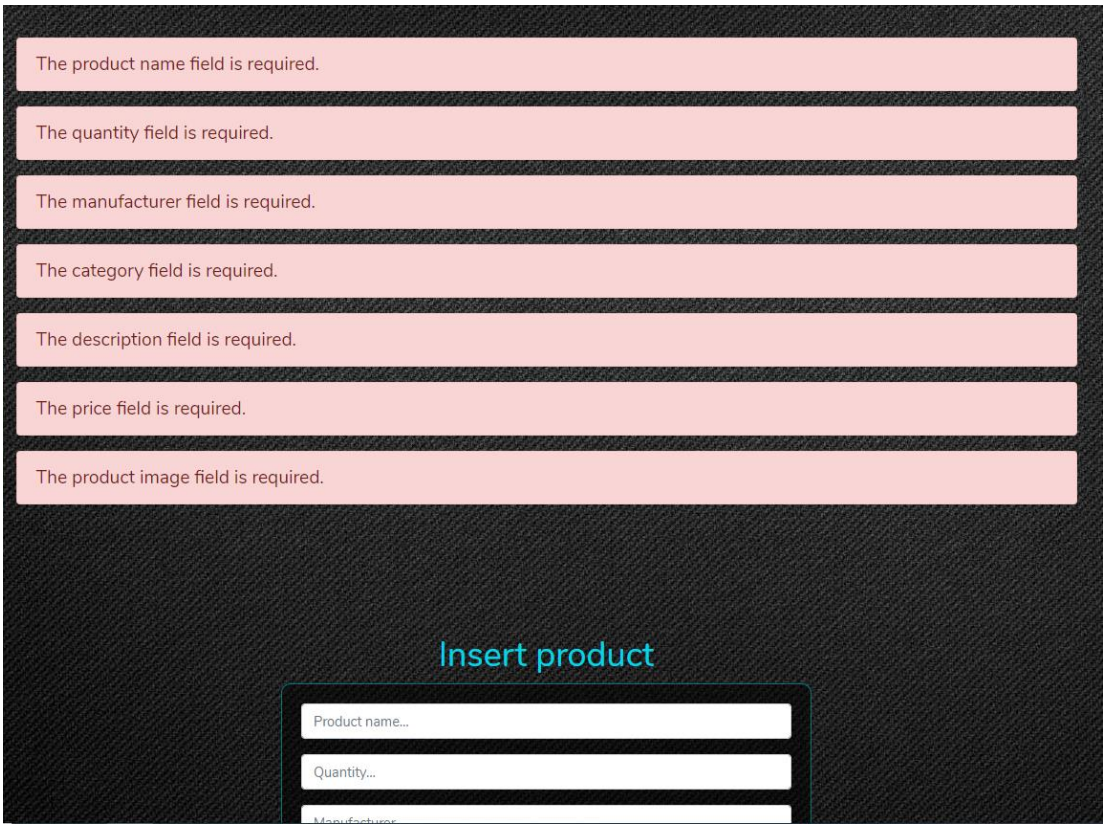


Слика 13. Фрома за додавање производа

Случајеви при којима се може појавити грешка:

- Покушај убацивања производа са неким празним пољима,
- Погрешан унос у одређеним пољима (рецимо ако корисник покуша да унесе текст у пољу за цену),
- Након попуњавања свих поља и додавања производа, може се јавити проблем са базом.

Након исписаних случајева тестер иде редом и проверава. У првом случају, тестер треба да провери да ли се може додати производ а да се притом не попуне поља. Провера и резултат се могу видети на Слици 14.



Слика 14. Резултат теста црном кутијом

Резултат теста: Ако корисник покуша да унесе производ, а притом није попунио неко од поља у форми, биће обавештен које поље није попуњено и тражиће му се да попуни то поље. Самим тим статус теста је позитиван и тестер може да настави са осталим случајевима.

8.3 Пример тестирања белом кутијом

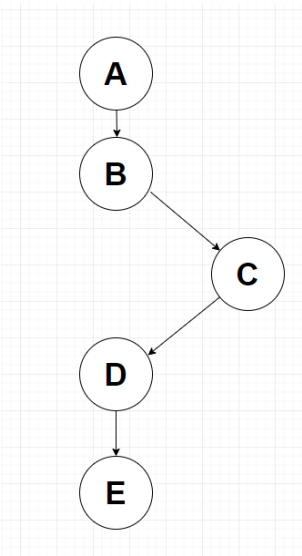
Пример тестирања белом кутијом ће бити приказан на истом делу апликације, као и тестирање црном кутијом, да би се упоредила разлика тестирања. Циљ тестирања код ове методе је да се изврше све програмске структуре (петље и блокови). Када се захтева од тестера да провери неки део кода, он добије само код, али не добија спецификацију шта тај код треба да ради и који резултат даје. Тако да је задатак тестера да проучи добијени код и да разуме шта свака линије кода ради и чему служи. Након проучавања, тестер треба да направи стабло редоследа извршавања сваког дела тог кода, притом, петља или упит су један део кода, блок кода (дефинисање променљивих, манипулисање промнљивима, као што је сабирање, одузимање,итд.) је други део, а функције (позивање неке функције која је унутар те класе или неке друге класе или рецимо позивање return-а) су трећи и сваки део кода се означава различитим словом (Слика 15).

```
public function store(Request $request)
{
    $this->validate($request,[
        'productName'=> 'required',
        'quantity'=> 'required',
        'manufacturer'=> 'required',
        'category'=> 'required',
        'description'=> 'required',
        'price'=> 'required',
        'productImage' =>'image|required|max:1999'
    ]);
    //Handle File Upload
    if($request->hasFile('productImage')){
        //Get filename with the extension
        $filenameWithExt = $request->file('productImage')->getClientOriginalName();
        // Get just filename
        $filename = pathinfo($filenameWithExt, PATHINFO_FILENAME);
        // Get just extension
        $extension = $request->file('productImage')->getClientOriginalExtension();
        // Filename to store
        $fileNameToStore = $filename.'_'.time().'.'.$extension;
        // Upload the image
        $path = $request->file('productImage')->storeAs('public/productImage', $fileNameToStore);
    }

    // Create Product
    $product = new Product;
    $product->productName = $request->input('productName');
    $product->quantity = $request->input('quantity');
    $product->manufacturer = $request->input('manufacturer');
    $product->category = $request->input('category');
    $product->description = $request->input('description');
    $product->price = $request->input('price');
    $product->productImage = $fileNameToStore;
    $product->save();

    return redirect('/products')->with('success','Product Inserted');
}
```

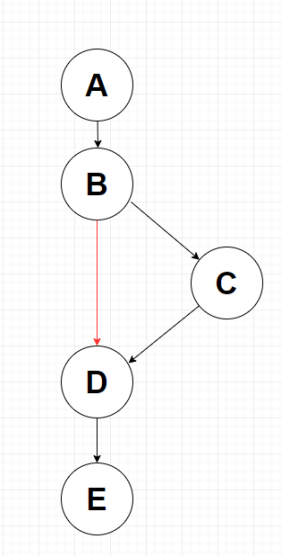
A
B
C
D
E

Слика 15. Део кода који тестер испитује методом беле кутије

Када се означе сви делови кода словима, креира се стабло редоследа извршавања и оно се може видети на Слици 16.



Слика 16. Првобитно стабло



Слика 17. Стабло са пронађеном грешком

Циљ овог тестирања је пронаћи део на стаблу код кога се из једног дела кода не одлази у други део кода. У овом случају може се видети да се из дела кода B у део кода D може доћи једино преко дела кода C, што представља грешку у коду, а директан пут из B у D није могућ (Слика 17). Анализом кода на Слици 15, та грешка је лако уочљива.. Ако се не би извршио део кода C, а то је све оно што је унутар if услова, онда се никада не би креирала променљива *$fileNameToStore* и то избацује грешку у програму. Та грешка се јавља када корисник не унесе никакву слику за производ и онда би програм стао, јер не може да ради са променљивом која не постоји. Тестирањем методом црне кутије није уочена никаква грешка у овом делу кода. Међутим, тестирањем методом беле кутије та грешка је лако уочена. Како не би дошло до прескакања грешака, неопходно је увек апликацију тестирати и методом беле и методом црне кутије, што заправо представља методу сиве кутије.

8.4 Пример јединичног тестирања

Јединично тестирање представља тестирање сваке компоненте апликације. Тестер треба да узме компоненту кода, да разуме код компоненте и да зна тачан излаз компоненте. Када све то зна, тестер пише код за тестирање компоненте. Пример овог тестирања је реализован коришћењем PHPUnit-а који је уграђен унутар Laravel развојног окружења.

PHPUnit је независно, бесплатно развојно окружење за јединично тестирање кодова апликација писаних у PHP програмском језику. Свака метода унутар скрипте, представља један тест, тако да назив сваке методе треба да што детаљније описује сврху теста. Тест се унутар Laravel-а креира помоћу наредбе *php artisan make:test NazivTesta*. Сваки креирани тест се налази у *tests* фолдеру. Покретање теста се може извршити помоћу наредбе *phpunit NazivTesta,* а резултат извршавања теста може бити *OK*, што говори да тест није пронашао никакву грешку, или *Error*, што говори да је тест пронашао грешку у коду коју је потребно исправити. Никако не треба назвати методу *тест1()* или *проба()*, већ је потребно описати детаљно шта се тестира, рецимо:

тестирањеДодавањаПроизводаОдСтранеАдмина().

Како би се назначило да те методе представљају тест, морају да имају префикс “тест” или да имају у коментару изнад методе анотацију *@test*. PHPUnit не може да покрене тест који је *private* или *protected*, тако да свака метода мора да буде *public*. [5]

Начин на који PHPUnit развојно окружење проверава апликацију је помоћу такозваних метода потврде (енгл. *assertions methods*). Оне се користе како би се проверили излази делова апликација. Неке од најкоришћенијих метода потврде су:

- assertEmpty ($result) – проверава да ли је резултат празан,
- assertEquals ($expected, $result) – проверава да ли је добијени резултат исти као и очекивани,

- assertGreaterThan (condition, $result) – проверава да ли је резултат већи од задатог услова (такође постоје и друге варијације ове функције: LessThan, GreaterThanOrEqual и LessThanOrEqual),
- assertContains (value, $result) – проверава да ли променљива има одређену вредност,
- assertType (type, $result) – проверава да ли је резултат задатог типа,
- assertNull ($result) – проверава да ли је вредност променљиве null,
- assertFileExists ($file) – проверава да ли фајл постоји,
- assertRegExp ($result, regularExpression) – проверава резултат са задатим регуларним изразом.
- assertTrue ($result) – проверава да ли је улаз тачан,
- assertFalse ($result) – проверава да ли је улаз нетачан.

Све горе наведене функције као излаз дају *True*, ако тест није наишао или *False* ако је тест наишао на грешку. Поред наведених постоји још метода потврде, али ове су најкоришћеније. Приликом писања тестова помоћу метода потврде, постоји једно правило "један тест, једна метода потврде". Ово правило говори да за сваки тест који се напише може се искористити само једна метода потврде. Неки програмери тврде да је то губљење времена и да треба што више метода потврде искористити унутар једног теста.

```
public function testIsMyString(){
    $string = "Mostly Harmless";
    $this -> assertGreaterThan(0, strlen($string));
    $this -> assertContains("42", $string);
}
```

Слика 18. Пример кршења правила "један тест, једна метода потврде"[6]

На слици 18 је пример кршења правила "један тест, једна метода потврде". У овом примеру се тестирају две различите ствари. Прво, проверава се да ли променљива *$string* садржи било какав карактер, тј да ли је дужина већа од нуле а одмах након тога проверава се да ли *$string* садржи број "42". Тест би рецимо био неуспешан, тј избацио би грешку, ако је вредност променљиве *$string* једнака "fortytwo", иако задовољава прву методу потврде. Због тога, тестер не може лепо да провери ту функционалност, тако да је најбоље решење да се напишу два теста са по једном методом потврде.

Свака метода унутар скрипте је засебна, тако да није битан редослед извршавања тестова, односно метода. У случају да тестер жели да једна метода зависи од друге, користи се анотација *@depends*. На слици 19 је приказан пример како се користи зависност функција. Фунцкија *testPush()* зависи од функције *testEmpty()*. тако што чека проверу промењиве *$stack*. Ако је тест позитиван, функција *testPush()* може да почне са извршавањем. Са слике се може видети, да је пре дефинисања функције *testPush()* додата анотација *@depends* од које методе зависи та метода.

```
public function testEmpty(){
    $stack = [];
    $this -> assertEmpty($stack);

    return $stack;
}

/** @depends testEmpty */
public function testPush(array $stack){
    array_push($stack,'foo');
    $this->assertSame('foo', $stackp[count($stack)-1]);
    $this->assertNotEmpty($stack);

    return $stack;
}
```

Слика 19. Пример коришћења зависности функција[6]

Тест метода може да прима и произвољне аргументе. Ови аргументи се добијају помоћу једне или више метода. Те методе се називају провајдер методе и означавају се *@dataProvider* анотацијом. Провајдер методе морају да буду *public* и да враћају низ низова или објекат који у себи садржи низ. Суштина ових метода је да другу методу снабдевају подацима. На слици 20 се може видети да је метода *additionProvider()* провајдер метода методи *testAdd()*. То се постиже додавањем анотације *@dataProvider* изнад методе *testAdd()*заједно са називом методе која је провајдер метода. У овом случају то је *additionProvider()*.

```php
/**@dataProvider additionProvider*/
public function testAdd($a, $b, $expected)
{
    $this->assertSame($expected, $a + $b);
}
public function additionProvider()
{
    return [
        [0, 0, 0],
        [1, 1, 3]
    ];
}
```

Слика 20. Пример провајдер методе[6]

У наставку ће бити приказано јединично тестирање над апликацијом. Први пример јединичног теста представља проверу ауторизације. Не може сваки корисник да приступи сваком делу апликације, тј. радник и добављач имају различита права. Не сме се никако дозволити да добављач може да приступи страници која приказује производе јер је то искључиво право запослених у магацију. На Слици 21. приказан је код за проверу да ли је страница за приказ производа заштићена.

```php
/** @test */
public function checkAuthenticationForProductsPage()
{
    $response = $this->get('/products')
        ->assertRedirect('/login');
}
```

Слика 21. Метода за проверу заштите страница додавања производа[6]

Функција покушава да приступи страници за приказ производа и уједно проверава да ли при том приступу корисник, који није пријављен, бива преусмерен на страницу за пријављивање, јер нема право приступа том делу апликације. Резултат покретања овог теста је приказан на слици 22.

```
Aca@DESKTOP-DA4592R MINGW64 ~/Desktop/magacin
$ vendor/bin/phpunit --filter checkAuthenticationForProductsPage
PHPUnit 7.5.15 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 539 ms, Memory: 20.00 MB

OK (1 test, 2 assertions)
```

Слика 22. Резултат теста за аутентикацију

Резултат теста је позитиван, што значи да је страница заштићена. Тачније, сваки покушај непријављеног корисника да приступи страници приказа производа, проузрокује преусмеравање на страницу за пријављивање.

Други пример јединичног тестирања је провера да ли се производ који радник унесе у форми за унос производа уписује у базу. Метода за проверу ове функције налази се на слици 23.

```php
/** @test */
public function addProductThroughTheForm(){

    $this->withoutExceptionHandling();

    $response = $this->get('/products/create',[
        'productName'=> 'Laptop',
        'quantity'=> '123',
        'manufacturer'=> 'Asus',
        'category'=> 'lap',
        'description'=> 'opasanlaptop',
        'price'=> '123',
        'productImage' =>'test.png'
    ]);
    $this->assertCount(1,Product::all());
}
```

Слика 23. Метода за проверу уношења производа у базу

Функција *withoutExceptionHandling()* није део тестирања, већ је ту да би се лакше пронашла грешка уколико се појави. PHPUnit развојно окружење аутоматски разрешава сваку грешку и избацјује статус код, а не каже где се та грешка појавила и шта је проузроковало. У наставку је потребно изабрати податке о производу, како би тест могао да покуша да дода производ. На крају се проверава да ли је тај поризвод убачен у базу. Резултат теста је приказан на слици 24.

```
Aca@DESKTOP-DA4592R MINGW64 ~/Desktop/magacin
$ ./vendor/bin/phpunit --filter addProductThroughTheForm
PHPUnit 7.5.15 by Sebastian Bergmann and contributors.

.                                                                   1 / 1 (100%)

Time: 554 ms, Memory: 20.00 MB

OK (1 test, 1 assertion)
```

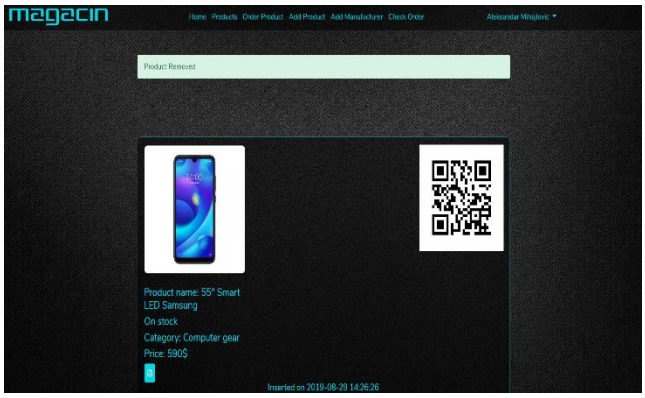
Слика 24. Резултат теста додавања производа

Резултат теста је позитиван: производ који радник дода се убацује у базу.

8.5 Пример интеграционог тестирања

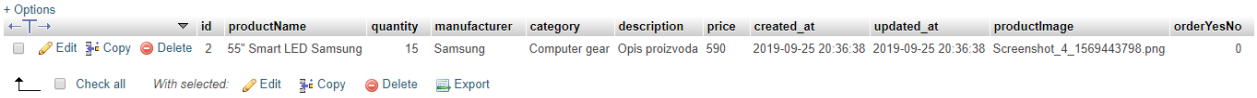
Интеграционо тестирање представља проверу да ли и како компоненте апликације функционишу као целина. Тестирање ће бити демонстрирано на примеру брисања производа. Ту се тестирају две компоненте: прва компонента је брисање производа на *front-end*-у, док је друга компонента брисање тог истог производа из базе. Код интеграционог тестирања се ове компоненте не тестирају посебно, јер је то урађено при јединичном тестирању и зато се увек апликација у потпуности тестира прво јединично, а затим интеграционо. Тестер покушава да притиском на дугме за брисање одређеног производа избрише производ (Слика 25.) и резултат притиска на *front-end*-у је приказан на слици 26.

Слика 25. Брисање поризвода



Слика 26. Резултат брисања производа

Може се закључити да је тест *front-end*-у успешан, тј. да нема грешака приликом брисања производа. Да би се проверило да ли је производ избрисан из базе, тестер мора да оде у базу података и да погледа да ли тај исти производ у њој постоји или не (Слика 27).



Слика 27. Табела производа у бази података

Пошто је проверено да се и у бази брише одабрани производ, тестер може да закључи да је тест успешан и на страни *back-end*-а.Самим тим и цео интеграциони тест је успешан. Након што тестер провери све интеграционе тестове у апликацији, мора да напише извештај о свим тестовима.

9. Закључак

Тестирање софтвера је процес који се користи да би се утврдила исправност, потпуност и квалитет развијеног софтвера. Самим тим овај процес представља и начин да се осигура мањи број грешака, мањи трошак одржавања и свеукупне цене софтвера. Током развоја софтвера, све се мора бити проверено више пута пре објаве производа. Да би остали у послу као ИТ компанија, императив је имати тим софтвер тестера. Производи са критичним баговима, могу проузроковати велике губитке у пословању. Зато данас, свака озбиљна ИТ компанија инвестира у свој тим тестера који имају задатак да до тога не дође. Ово је уједно и разлог зашто су софтвер тестери данас тражени на тржишту. Из тог разлога, тестирање данас постаје стандард који никако не сме да се занемари.

10. Литература

[1]. Живковић М. (2018). *Тестирање софтвера*. Београд: Универзитет Сингидунум .

[2]. Software Testing Tutorial: https://www.guru99.com/software-testing.html (датум приступа страници: 4.9.2019.)

[3]. What Is Software Testing – Definition, Types, Methods, Approaches: https://www.softwaretestingmaterial.com/software-testing/ (датум приступа страници: 5.9.2019.)

[4]. Introduction – Laravel – The PHP Framework For Web Artisans: https://laravel.com/docs/4.2/introduction (датум приступа 21.09.2019)

[5]. PHPUnit – The PHP Testing Framework: https://phpunit.de/ (датум приступа страници: 22.9.2019.)

[6]. PHPUnit Manual – PHPUnit 8.3 Manual https://phpunit.readthedocs.io/en/8.3/ (датум приступа страници: 3.10.2019)